An Integrated Hardware-Software Approach to Flexible Transactional Memory*

Arrvindh Shriraman, Michael F. Spear, Hemayet Hossain, Virendra J. Marathe, Sandhya Dwarkadas, and Michael L. Scott

Department of Computer Science, University of Rochester
{ashriram,spear,hossain,vmarathe,sandhya,scott}@cs.rochester.edu

ABSTRACT

There has been considerable recent interest in both hardware and software transactional memory (TM). We present an intermediate approach, in which hardware serves to accelerate a TM implementation controlled fundamentally by software. Specifically, we describe an alert on update *mechanism (AOU) that allows a thread to receive fast, asynchronous notification when previously-identified lines are written by other threads, and a* programmable data isolation *mechanism (PDI) that allows a thread to hide its speculative writes from other threads, ignoring conflicts, until software decides to make them visible. These mechanisms reduce bookkeeping, validation, and copying overheads without constraining software policy on a host of design decisions.*

We have used AOU and PDI to implement a hardware-accelerated software transactional memory system we call RTM. We have also used AOU alone to create a simpler "RTM-Lite". Across a range of microbenchmarks, RTM outperforms RSTM, a publicly available software transactional memory system, by as much as 8.7× (geometric mean of 3.5×) in single-thread mode. At 16 threads, it outperforms RSTM by as much as 5×, with an average speedup of 2×. Performance degrades gracefully when transactions overflow hardware structures. RTM-Lite is slightly faster than RTM for transactions that modify only small objects; full RTM is significantly faster when objects are large. In a strong argument for policy flexibility, we find that the choice between eager (first-access) and lazy (commit-time) conflict detection can lead to significant performance differences in both directions, depending on application characteristics.

Categories and Subject Descriptors: B.3.2 [Memory Structures]: Design Styles—Shared memory D.1.3 [Programming Techniques]: Concurrent Programming—Parallel programming C.1.2 [Processor Architectures]: Multiprocessors

General Terms: Performance, Design, Languages

Keywords: Transactional memory, Cache coherence, Multiprocessors, RSTM

*This work was supported in part by NSF grants CCR-0204344, CNS-0411127, CNS-0615139, and CNS-0509270; an IBM Faculty Partnership Award; equipment support from Sun Microsystems Laboratories; and financial support from Intel and Microsoft.

ISCA'07, June 9–13, 2007, San Diego, California, USA.

1. INTRODUCTION AND BACKGROUND

Transactional memory (TM) has emerged as a promising alternative to lock-based synchronization. TM systems seek to increase scalability, reduce programming complexity, and overcome the semantic problems of deadlock, priority inversion, and non-composability associated with locks. Originally proposed by Herlihy and Moss [9], TM borrows the notions of atomicity, consistency, and isolation from database transactions. In a nutshell, the programmer or compiler labels sections of code as *atomic* and relies on the underlying system to ensure that their execution is serializable and as highly concurrent as possible. Several hardware [1, 3, 7, 14, 16, 18–20] and software [5, 8, 12, 22, 24] TMs have been proposed. Hardware has the advantage of speed, but embeds significant policy in silicon. Software can run on stock processors and preserves policy flexibility, but incurs significant overhead to track data versions, detect conflicts between transactions, and guarantee a consistent view of memory.

We propose that hardware serve simply to optimize the performance of transactions that are controlled fundamentally by software. We present a system, RTM, that embodies this philosophy. The RTM software (currently based on a modified version of the RSTM software TM [13]) retains policy flexibility, and implements transactions unbounded in space and in time.

The RTM hardware consists of 1) an *alert-on-update* mechanism (AOU) for fast software-controlled conflict detection; and 2) *programmable data isolation* (PDI), which allows potentially conflicting readers and writers to proceed concurrently under software control. AOU is the simpler and more general of the mechanisms. It can be used for almost any task that benefits from fine-grain access control. In RTM, it serves to capture transaction conflicts and guarantee memory consistency without the heavy cost of continually *validating* objects that were previously read [27]. PDI additionally eliminates the cost of data copying or logging in bounded transactions. In our experiments we evaluate both full RTM (RTM-F) and an "RTM-Lite" that uses only AOU.

Damron et al. [4] describe a design philosophy for a hybrid TM system in which hardware makes a "best effort" attempt to complete transactions, falling back to software when necessary. The goal is to leverage almost any reasonable hardware implementation. Kumar et al. [10] describe a specific hardware–software hybrid that builds on the software system of Herlihy et al. [8]. Unfortunately, this system still embeds significant policy in silicon. It assumes, for example, that conflicts are detected as early as possible, disallowing either read-write or write-write sharing. Scherer et al. [12, 23] report performance differences across applications of 2×–10× *in each direction* for this design decision, and for contention management and metadata organization.

By leaving policy to software, AOU and PDI allow us to experiment with alternative policies on central TM issues such as data granularity (e.g., word v. object-based), metadata organization, progress guarantees (blocking/nonblocking), conflict detection, contention management, nesting, privatization, and virtualization. We focus in this paper on conflict detection: we permit, but do not require, read-write and write-write sharing, with delayed detection of conflicts. We also employ a software *contention manager* [23] to arbitrate conflicts and determine the order of commits.

Like the Damron and Kumar hybrid proposals, RTM falls back to a software-only implementation of transactions in the event of overflow. Because conflicts are handled in software, speculatively written data can be made visible at commit time by the local cache, with no need for global coordination in hardware. Moreover, these speculative writes (and a small amount of nonspeculative metadata) are *all* that must remain in the cache for fast-path execution: data that were speculatively *read* or *nonspeculatively* written can safely be evicted at any time. Hence, in contrast not only to the hybrid proposals, but also to TLR [19], LTM [1], VTM [20], and LogTM [16], it can accommodate "fast path" execution of significantly larger transactions with a given size of cache. Nonspeculative loads and stores are permitted in the middle of transactions—in fact they constitute the hook that allows us to implement policy in software.

We describe an implementation of AOU and PDI that can be integrated into either the L1 level of a CMP with a shared L2 cache, or the L2 level of an SMP with write-through L1 caches. We describe an implementation based on the classic snoop-based MESI protocol. Other implementations (for directory-based protocols) are a subject of ongoing work. Likewise, while our current software inherits a variety of policies (in particular, nonblocking semantics and object-level granularity) from RSTM, our hardware could be used with a variety of other software TMs, including systems that track conflicts at word granularity or use locks to make updates in place.

RTM was originally introduced (without performance results) in a 2005 technical report and a paper at TRANSACT'06 [26]. Researchers in the McRT group at Intel subsequently published a variant of AOU that uses synchronous polling instead of asynchronous events to detect cache line evictions. Their HASTM system [21] resembles RTM-Lite in its emphasis on software control, but employs a more streamlined software stack, with blocking semantics, eager-only conflict detection, simplified contention management, and weak guarantees of correctness.

For a suite of microbenchmarks with varying access patterns, RTM improves the performance of common-case bounded transactions by an average (geometric mean) of 3.5× relative to RSTM on one thread, while retaining RSTM's good scalability as the number of threads increases. Using AOU alone, RTM-Lite is able to dramatically reduce the overhead of validation. It loses to full RTM, however, for transactions that modify large objects. When transactions overflow the available hardware support, performance degrades linearly with the fraction of overflowed transactions. The choice between eager (first access) and lazy (commit time) detection of conflicts, enabled by software control of PDI, can result in differences in performance in either direction depending on the application access pattern (up to two orders of magnitude in one of our microbenchmarks), demonstrating the need for policy flexibility.

Section 2 describes our hardware mechanisms in detail, including instruction set extensions, coherence protocol, and the mechanism used to detect conflicts and abort remote transactions. Section 3 then describes the RTM runtime that leverages this hardware support. Section 4 evaluates the performance of RTM in comparison to coarse-grain locks, RSTM, and the "RTM-Lite" system, which uses AOU but not PDI. It also presents results to demonstrate the benefits of policy flexibility. We conclude in Section 5 with a summary of contributions and future directions.

2. RTM HARDWARE

2.1 Alert-On-Update (AOU)

AOU facilitates conflict detection by selectively exposing coherence events (potential writes by other processors) to user programs: threads register an *alert handler* and then mark selected lines as *alert-on-update*. When a marked line is lost from the cache, the cache controller notifies the local processor, effecting a spontaneous subroutine call to the handler of the current thread. The handler is informed of the nature of the event (and potentially of the address of the affected line, though we do not use this information/feature in RTM). Because a line may be lost due not only to coherence but to conflict or capacity misses, a handler must in general double-check the cause of the alert.

Registers	
`%aou_handlerPC`:	address of handler to be called on a user-space alert
`%aou_oldPC`:	PC immediately prior to call to `%aou_handlerPC`
`%aou_alertType (2bits)`:	remote_write, lost_alert, or capacity/conflict eviction
`%alert_enable (1bit)`:	set if alerts are to be delivered; unset when they are masked
interrupt vector table	one extra entry to hold address of handler for kernel-mode alerts
Instructions	
`set_handler %r`	move `%r` into `%aou_handlerPC`
`clear_handler`	clear `%aou_handlerPC` and flash-clear alert bits for all cache lines
`aload %r`	set alert bit for cache line containing the address in `%r`; set overflow condition code to indicate whether the bit was already set
`arelease %r`	unset alert bit for line containing the address in `%r`
`arelease_all`	flash-clear alert bits on all cache lines
`enable_alerts`	set the alert-enable bit
Cache	
one extra bit per line, orthogonal to the usual state bits	

Table 1: Alert-on-update hardware requirements.

Implementation of AOU relies on the cache coherence protocol, but is essentially independent of protocol details. Table 1 summarizes hardware requirements. These include special registers to hold the address of the user-mode handler and a description of the current alert; an extra entry in the interrupt vector table (for alerts that happen while running in kernel mode); and instructions to set and unset the user-mode handler and to mark and unmark cache lines (i.e., to set and clear their alert bits).

*ALoad*s serve two related roles in RTM, which we describe in more detail in Section 3. First, every transaction *ALoad*s a location that describes its current status. If any other transaction aborts it (by modifying this location), the first transaction is guaranteed to notice. Second, a transaction can *ALoad* a word of metadata associated with a given object. If writers modify that word before committing changes to the object, readers are guaranteed to notice. Via this mechanism, we permit lazy conflict detection (i.e., we do not require that conflicts be detected as soon as some word of the object is speculatively written) by controlling when the metadata is *ALoad*ed and when it is written with respect to the rest of the object.

2.2 Programmable Data Isolation

Caches inherently provide data buffering, but coherence protocols normally propagate modifications quickly to all copies. As in

Registers

`%t_in_flight`:	a bit to indicate that a transaction is currently executing
`%hardware_t`:	a bit to indicate that `tload` and `tstore` should be treated as such

Instructions

`begin_t`	set the `%t_in_flight` bit to indicate the start of a transaction
`begin_hw_t`	set the `%hardware_t` and `%t_in_flight` bits to indicate a transaction in which `tloads` and `tstores` should be treated as such
`tstore [%r1], %r2`	write the value in register `%r2` to the word at address `%r1`; isolate the line (*TMI* state)
`tload [%r1], %r2`	read the word at address `%r1`, place the value in register `%r2`, and tag the line as transactional
`abort`	discard all isolated (*TMI* or *TI*) lines; clear all transactional tags and reset the `%t_in_flight` and `%hardware_t` bits
`cas-commit [%r1], %r2, %r3`	compare `%r2` to the word at address `%r1`; if they match, commit all isolated writes (*TMI* lines) and store `%r3` to the word; otherwise discard all isolated writes; in either case, clear all transactional tags, discard all isolated reads (*TI* lines), and reset the `%t_in_flight` and `%hardware_t` bits

Cache

two extra stable states, TMI and TI, for isolated reads and writes; transactional tag for the MES states

Table 2: TMESI hardware requirements.

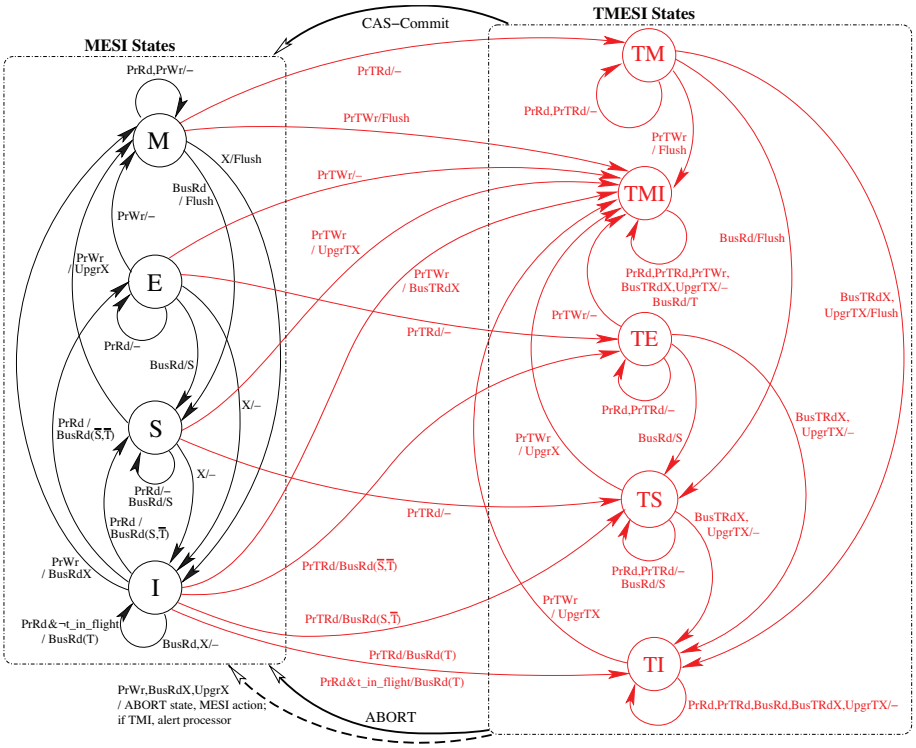


Figure 1: TMESI Protocol

Dashed boxes enclose the MESI and TMESI subsets of the state space. On a *CAS-Commit*, *TM*, *TE*, *TS*, and *TI* revert to *M*, *E*, *S*, and *I*, respectively; *TMI* reverts to *M* if the CAS succeeds, or to *I* if it fails. Notation on transitions is conventional: the part before the slash is the triggering message; after is the ancillary action ('–' means none). X stands for the set {BusRdX, UpgrX, BusTRdX, UpgrTX}. "Flush" indicates writing the line to the bus. S and T indicate signals on the "shared" and "threatened" bus lines respectively. Plain, they indicate assertion by the local processor; parenthesized, they indicate the signals that accompany the response to a BusRd request. An overbar means "not signaled". For simplicity, we assume that the base protocol prefers memory–cache transfers over cache–cache transfers. The dashed transition from the TMESI state space to the MESI state space indicates that actions occur only on the corresponding cache line. "ABORT state" is the state to which the line would transition on abort. The solid "CAS-Commit" and "ABORT" transitions from the TMESI state space to the MESI state space operate on all transactional lines.

most hardware TM proposals [1, 7, 16, 20], we allow a thread to delay this propagation while executing speculatively, and then to make an entire set of changes visible to other threads atomically. We use a level of cache close to the processor to hold the new copy of data, and rely on shared lower levels of the memory hierarchy to hold the old copy. Unlike most other hardware TM designers, however, we allow software to limit the extent to which transactionally read and written lines are visible to the coherence protocol, allowing those lines to be read and written transactionally even when they are also being written by some other, concurrent transaction.

We describe an implementation based on the traditional snoop-based MESI coherence protocol, which we label TMESI. Table 2 summarizes hardware requirements. Figure 1 presents the state transition diagram. Potentially speculative reads and writes use *TLoad* and *TStore* instructions. These instructions are interpreted as speculative when the hardware transaction bit (`%hardware_t`) is set (and an alert handler is in place). As described in Section 3, this allows the same code path to be used by both fast-path transactions and those that overflow the available hardware support. *TStore* is used for writes that require isolation. *TLoad* is used for reads that can safely be cached despite remote *TStore*s.

Speculative reads and writes employ two new coherence states: *TI* and *TMI*. These states allow a software policy, if it chooses, to perform lazy detection of read-write and write-write conflicts. Hardware helps in the detection task by piggybacking a *threatened* (T) signal/message, analogous to the traditional shared (S) signal/message, on responses to read requests whenever the line exists in *TMI* state somewhere in the system. The T signal warns a reader of the existence of a potentially conflicting writer.

TMI serves to buffer speculative local writes. Regardless of previous state, a line moves to *TMI* in response to a PrTWr (the result of a *TStore*), tagging any necessary coherence message as a transactional access. A *TMI* line then reverts to *M* on commit and to *I* on abort. Software must ensure that among any concurrent con-

flicting writers, at most one commits, and if a conflicting reader and writer both commit, the reader serializes first. The first *TStore* to a modified cache line results in a writeback prior to transitioning to *TMI* to ensure that lower levels of the memory hierarchy have the latest non-speculative value. A line in *TMI* state threatens read requests and suppresses its data response, allowing lower levels of the hierarchy to supply the non-speculative version of the data.

TI allows continued use of data that have been read by the current transaction, but that may have been speculatively written by a concurrent transaction in another thread. An *I* line moves to *TI* when threatened during a *TLoad*(speculative caching of ordinary loads is also allowed inside a transaction so long as software conventions detect the read-write conflict); an *M*, *E*, or *S* line that is tagged transactional (indicating that the current transaction has previously performed a *TLoad*) moves to *TI* when written by another processor. A *TI* line must revert to *I* when the current transaction commits *or* aborts, because a remote processor has made speculative changes which, if committed, would render the local copy stale. No writeback or flush is required since the line is not dirty. Even during a transaction, silent eviction and re-read of a *TI* or transactional *M*, *E*, or *S* line is not a problem; software ensures that no writer can commit unless it first aborts the reader. A non-transactional store (whether from the local processor or from a remote processor via an exclusive request) to a line in transactional state reverts the line to the state it would be in if aborted, and then performs the corresponding MESI coherence protocol action. In addition, if the line was in *TMI* state, an alert is triggered.

The *CAS-Commit* instruction performs the usual function of compare-and-swap. In addition, if the CAS succeeds, *TMI* lines revert to *M*, making their data visible to other readers through normal coherence actions. If the CAS fails, *TMI* lines are invalidated, and software branches to an abort handler. In either case, *TI* lines revert to *I* and any transactional tags are flashed clear on *M*, *E*, and *S* lines. The motivation behind *CAS-Commit* is simple: software TM systems invariably use a CAS or its equivalent to commit the current transaction; we overload this instruction to make buffered transactional state once again visible to the coherence protocol. The *Abort* instruction clears the transactional state in the cache in the same manner as a failed *CAS-Commit*.

To the best of our knowledge, RTM and TCC [7] are the only hardware or hybrid TM systems that permit read-write and write-write sharing; other schemes all perform eager conflict detection at the point where a conventional coherence protocol must invalidate a speculatively read line or demote a speculatively written line. By allowing a reader transaction to commit before a conflicting writer, RTM permits significant concurrency in the face of long-running writers. Write-write sharing is more problematic, since it can't result in more than one commit. Even so, it allows us to avoid aborting a transaction in favor of a competitor that is ultimately unable to commit; it may also be desirable in conjunction with *early release* [12]. Note that nothing about the TMESI protocol *requires* read-write or write-write sharing; if the software protocol detects and resolves conflicts eagerly, the *TI* state will simply go unused.

2.3 Ordering and Atomicity Requirements

On machines with relaxed memory models, `begin_t` and `begin_hw_t` must be acquire fences, and `cas-commit` must be a release fence. An `aload` by itself is not a fence. As used in RTM, however, both `aload` (and `enable_alerts`) must precede subsequent *TLoad*s and *TStore*s; this can be arranged with an explicit acquire fence.

Since alerts are only hints (they are never elided, but may be spurious), it isn't essential to resolve prior branches and exceptions prior to an `aload`, but resolution would be required prior to `enable_alerts`. Alternatively, `aload` (and `enable_alerts`) could take effect at retirement (similar to stores), with an alert delivered if the line is invalid at the time of retirement.

Simultaneous transactional and non-transactional stores to the same cache line within a transaction are not allowed (and may result in loss of transactional modifications to the line, triggering an alert, which the software can use to abort the transaction). Non-transactional stores concurrent with executing transactions cause the transaction to receive an alert if it has the line in *TMI* state; they invalidate the line if in other transactional states. RTM leverages this behavior in its handling of memory management metadata, which is colocated (in the same cache line) with transactional object data: updates to this metadata are made only outside transactions, thereby preserving coherence.

To simplify the management of transactional metadata, our implementation of RTM employs a *Wide-CAS* instruction (not shown in Table 2) that implements compare-and-swap across multiple contiguous locations (within a single cache line). As in Itanium's `cmp8xchg16` instruction, if the first two words at location A match their "old" values, all words are swapped with the "new" values (loaded into contiguous registers). Success is detected by comparing old and new values in the registers.

3. RTM SOFTWARE

The RTM runtime is based on the open-source RSTM system [13], a C++ library that runs on legacy hardware. RTM uses alert-on-update and programmable data isolation to reduce bookkeeping and validation overheads, and to avoid copying, thereby improving the speed of "fast path" transactions. When a transaction's execution time exceeds a single quantum, or when the working set of a transaction exceeds the *ALoad* or *TStore* capacity of the cache, RTM restarts the transaction in a more conservative "overflow mode" that supports unboundedness in both space and time.

3.1 The RTM API

RTM runs the same source code—and supports the same programming model—as RSTM. Transactions are lexically scoped, and delimited by `BEGIN_TRANSACTION` and `END_TRANSACTION` macros. The first of these sets the alert handler for a transaction and configures per-transaction metadata. The second issues a *CAS-Commit*.

Objects accessed transactionally must derive from the RTM `Object` class. This class contains metadata described in Section 3.2, and provides access to transaction-safe memory management routines, which defer the reuse of deleted space until we are certain that no doomed transaction retains a dangling reference.

In order to access fields of an object, a thread must obtain read or write permission by performing an `open_RO` or `open_RW` call. The API also provides a `release` method [8], which allows a programmer with application-specific semantic knowledge to inform the runtime that conflicts on the object are no longer cause to abort the transaction. `Release` is a potentially unsafe optimization, which must be used with care.

The runtime sometimes requires that a set of related metadata updates be allowed to complete, i.e., that the transaction not be aborted immediately. This is accomplished by setting a "do not abort me" flag. If an alert occurs while the flag is set, the handler defers its normal actions, sets another flag, and returns. When the runtime finishes its updates, it clears the first flag, checks the second, and jumps back to the handler if action was deferred. This

"deferred abort" mechanism is also available to user applications, where it serves as a cheap, non-isolated approximation of open nested transactions [17].

3.2 Metadata

Every RTM transaction is represented by a *descriptor* (Figure 2) containing a serial number and a word that indicates whether the transaction is currently `ACTIVE`, `COMMITTED`, or `ABORTED`. The serial number is incremented every time a new transaction begins. It enables the reuse of descriptors without the need for cleanup in the wake of a successful commit.

Every transactional object is represented by a *header* containing five fields: a pointer to an "owner" transaction, the owner's serial number, pointers to valid (*old*) and speculative(*new*) versions of the object, and a bitmap listing overflow transactions currently reading the object (threads that need to run in overflow mode compete for one of 32 global overflow IDs).

If the serial numbers of an object header and descriptor do not match, then the descriptor's status is assumed to be `COMMITTED`. `ABORTED` overflow transactions must clean up all references to their descriptors before they start a new transaction. The memory manager allocates every object, header, and descriptor at a new cache-line boundary. On a system with large cache lines, internal fragmentation could be reduced (for headers and descriptors) by implementing A tags at sub-line granularity, as in HASTM [21]; we do not consider this option here.

`Open_RO` returns a pointer to the most recently committed version of the object. Typically, the owner/serial number pair indicates a `COMMITTED` transaction, in which case the New pointer is valid if it is not NULL; otherwise the Old pointer is valid. If the owner/serial number pair indicates an `ABORTED` transaction, then the Old pointer is always valid. As an optimization, `open_RO` also CASes a NULL pointer into the owner/serial number pair in order to avoid having to look up the status of the transaction descriptor multiple times per successful transaction. When the owner is `ACTIVE`, there is a conflict. An object never has entries in the overflow readers list while there is an `ACTIVE` owner.

`Open_RW` returns a pointer to a writable copy of the object. For fast-path transactions this is the valid version that would be returned by `open_RO`; updates will be buffered in the cache. For overflow transactions it is a *clone* or copy of the valid version.

At some point between its `open_RW` and commit time, a transaction must *acquire* every object it has written. The acquire operation first gets permission from a software *contention manager* [8, 23] to abort all transactions in the overflow reader list. It then writes the owner's ID, the owner's serial number, and the addresses of both the last valid version and the new speculative version into the header using *Wide-CAS*. Finally, it aborts any transactions in the overflow reader list of the freshly acquired object.[1]

At the end of a transaction, a thread issues a *CAS-Commit* to change its state from `ACTIVE` to `COMMITTED`. If the CAS fails because another thread has set the state to `ABORTED`, the transaction is retried.

3.3 Policy Flexibility

In a typical hardware TM system, contention management is performed by the cache controller that owns the line, based on the limited information available to it, while the requesting thread blocks waiting for a response. In RTM, contention management is performed by nonblocking software, executed by the thread that discovers the conflict, using whatever information the runtime designer wants.

Two transactions conflict only if they access the same object and at least one of them attempts to write it. In RTM this conflict is not visible until the writer *acquires* the object. Under *eager* conflict detection, acquisition occurs at open time, and read-write and write-write sharing are precluded. A writer aborts any extant readers, and once there is a writer, subsequent readers and writers must abort the eager writer before they can access the object. In contrast, under *lazy* conflict detection, acquisition is deferred until commit time, and read-write and write-write sharing are permitted.

Eager acquisition avoids the need to maintain an explicit list of written objects, making it faster for workloads in which aborts are rare. It may also avoid useless work by aborting a doomed transaction early. Lazy acquisition, by contrast, avoids the possibility of aborting a transaction in favor of a competitor that is subsequently unable to commit. It also allows a writer and one or more concurrent readers to *all* commit, so long as the readers do so first.

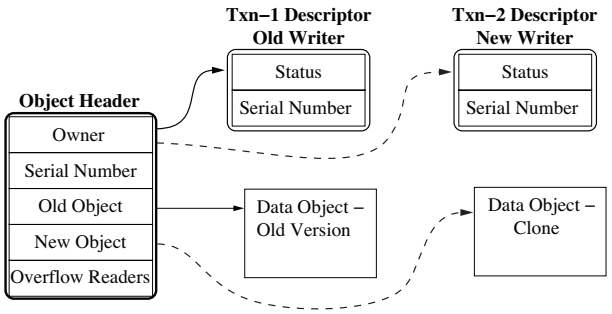
3.4 Fast-Path RTM Transactions

Eliminating Data Copying. A fast-path transaction calls `begin_hw_t` inside the `BEGIN_TRANSACTION` macro. Subsequent *TStore*s will be buffered in the cache; their data will remain invisible to other threads until the transaction commits (at the hardware level, of course, the existence of lines to which *TStore*s have been made is visible in the form of "threatened" signals/messages). As noted in Section 3.2, `open_RW` returns a pointer to the current version of an object when invoked by a fast-path transaction, thereby enabling in-place updates. Programmable data isolation thus avoids the need to create a separate writable copy, as is common in software TM systems (RSTM among them). When a fast-path transaction acquires an object, it writes a NULL into the New pointer, since the old pointer is both the last and next valid version. As a result, when a fast-path transaction aborts, it does not need to clean up the Owner pointers in objects it has acquired; since the owner has been working directly on the Old version of the data, a newly arriving transaction that sees mis-matched serial numbers will read the appropriate version.

Reducing Bookkeeping and Validation Costs. In most software TM systems, a transaction may be doomed to fail (because of conflicting operations in committed peers) well before it notices its fate. In the interim it may read versions of objects that are mutually inconsistent. This possibility raises the problem of *validation*: a transaction must ensure that inconsistency never causes it to perform erroneous operations that cannot be rolled back. In general, a transaction must verify that all its previously read objects are still valid before it performs any dangerous operation. Such validation can be a major component of the cost of software TM [27]: making readers visible to writers requires metadata updates that induce large numbers of cache misses; leaving them invisible leads to $O(n^2)$ total cost for a transaction that reads n objects.

ALoad allows validation to be achieved essentially for free. Whenever an object is read (or opened for writing with lazy acquire), the transaction uses *ALoad* to mark the object's header in the local cache. Since transactions cannot commit changes to an object without modifying the object header first, the remote acquisition of a locally *ALoad*ed line results in an immediate alert to the reader transaction. Since the header must be read in any case, the *ALoad* induces no extra overhead. Freed of the need to explicitly validate previously opened objects, software can also avoid the bookkeeping

[1] It is possible for a reader to enter the list after the acquirer finishes requesting permission to abort readers. In this case the late-arriving reader may be aborted without arbitration, ensuring correctness but not fairness.



Here a writer transaction is in the process of acquiring the object, overwriting the Owner pointer and Serial Number fields, and updating the Old Object pointer to refer to the previously valid copy of the data. A fast-path transaction will set the New Object field to null; an overflow transaction will set it to refer to a newly created clone. Several overflow transactions can work concurrently on their own object clones prior to *acquire* time, just as fast-path transactions can work concurrently on copies buffered in their caches.

Figure 2: RTM metadata structure.

overhead of maintaining those objects on a list. Best of all, perhaps, a transaction that acquires an object implicitly aborts all fast-path readers of that object simply by writing the header: fast-path readers need not add themselves to the list of readers in the header, and the $O(t)$ cost of aborting the readers is replaced by the broadcast invalidation already present in the cache coherence protocol.

Alerts are masked while running in the handler; before returning, the handler unmasks them with an `enable_alerts` instruction. If more than one alert occurs while masked, the hardware combines them into a single lost_alert event. An RTM transaction aborts and retries in overflow mode in response to such an event, or to invalidation or eviction of an A-tagged or *TMI* line.

Single Thread Optimizations. Metadata management and bookkeeping in API calls comprise a significant fraction of the overhead of software-managed TM, relative to coarse-grain locks. Much of this overhead can be elided for transactions that are guaranteed to run in isolation (i.e., when no other transaction is active in the system, as is the case when using a single thread). We employ a global flag to detect this situation. In single-thread mode, a fast-path transaction optimistically buffers all writes with PDI, while `Aloading` only its descriptor. It skips all bookkeeping associated with `open_RO` and `open_RW` calls. To begin a concurrent transaction, other threads must first examine the flag and abort the single-thread-mode transaction by writing to its descriptor. The aborted transaction will receive an alert and retry in conventional fast-path mode. This optimization is completely safe and maintains non-blocking progress since PDI guarantees that no intermediate writes are seen and AOU guarantees immediate abort.

3.5 Overflow RTM Transactions

Like most hardware TM proposals, fast-path RTM transactions are bounded in space and time. They cannot *ALoad* or *TStore* more lines than the cache can hold, and they cannot execute across a context switch, because we do not (currently) associate transaction IDs with tagged lines in the cache. To accommodate larger or longer transactions, RTM employs an *overflow* mode with only one hardware requirement: that the transaction's *ALoad*ed descriptor remain in the cache whenever the transaction is running. Since committing a fast-path writer updates written objects in-place, we must ensure that a transaction in overflow mode also notices immediately when it is aborted by a competitor. We therefore require that every transaction *ALoad* its own descriptor. If a competitor CAS-es its status to `ABORTED`, the transaction will suffer an immediate alert, avoiding the possibility that it will read mutually inconsistent data from within a single object.

Disabling Speculative Loads and Stores. In principle, a transaction that exceeds the capacity of the cache could continue to use the available space for as many objects as fit. For the sake of simplicity we do not currently pursue this possibility. Rather, a transaction that suffers a "no more space" alert aborts and retries in overflow mode. In this mode it leaves the `%hardware_t` bit clear, instructing the hardware to interpret *TLoad* and *TStore* instructions as ordinary loads and stores. This convention allows the overflow transaction to execute the exact same user code as fast-path transactions; there is no need for a separate version. An overflow transaction does, however, invoke `begin_t` in the `BEGIN_TRANSACTION` macro to indicate an active transaction.

Without speculative stores, `open_RW` calls in the overflow transaction must clone to-be-written objects. At acquire time, the WCAS instruction writes the address of the clone into the New field of the metadata. When `open_RO` calls encounter a header whose last Owner is committed and whose New field is non-null, they return the New version as the current valid version.

Limiting *ALoad*s. Though an overflow transaction cannot *ALoad* every object header it reads, it still *ALoad*s its own descriptor. It also writes itself into the Overflow Reader list of every object it reads; this ensures it will be explicitly aborted by writers.

While only one *ALoad*ed line is necessary to ensure immediate aborts and to handle validation, using a second *ALoad* can improve performance when a fast-path transaction and an overflow transaction are concurrent writers. If the overflow writer is cloning an object when the fast-path writer commits, the clone operation may return an internally inconsistent object. If the overflow transaction becomes a visible reader first, the problem is avoided. It is simpler, however, to *ALoad* the header and then clone the object. An acquire by another writing transaction will result in the clone operation suffering an alert. We assume in our experiments that the hardware is able (with a small victim cache) to prefer non-*ALoad*ed lines for eviction, and to keep at least two in the cache.

Context Switches. To support transactions that must be preempted, we require two actions from the operating system. When it swaps a transaction out, the operating system flash clears all the A tags. In addition, for transactions in fast-path mode, it executes the `abort` instruction to discard isolated lines. When it swaps the transaction back in, it starts execution in a software-specified *restart handler* (separate from the alert handler). The restart handler aborts and retries if the transaction was in fast-path mode or was swapped out in mid-clone; otherwise it re-*ALoad*s the transaction descriptor and checks that the transaction status has not been changed to `ABORTED`. If this check succeeds, control returns as normal; otherwise the transaction jumps to its abort code.

3.6 Privatization

Hardware TM systems typically provide a *strongly isolated* (a.k.a. strongly atomic [2]) programming model, in which the internal state of a transaction is isolated not only from other transactions, but from individual (nontransactional) loads and stores. We consider this model to be overkill: isolated accesses that are not ordered with respect to transactions constitute data races that are likely to be bugs [11, Section 2.1.2].

RTM does provide serialization between transactional and nontransactional accesses to the same location, however, for programs that are free of such races. As an example, our largest benchmark application, a locally-constructed implementation of Delaunay triangulation, has lengthy phases in which edges in the triangulation graph are partitioned among threads geometrically; these are separated by barriers from phases in which edges are protected by transactions. In a similar vein, our LinkedList and RBTree benchmarks employ *privatizing transactions* that remove an element from a transactional set, after which it can be accessed with ordinary loads and stores.[2]

Privatization introduces a pair of implementation challenges for STM: (1) a thread that privatizes data must see all changes made to that data by previously committed transactions; (2) a doomed transaction that continues, temporarily, to use data that has been privatized by another thread must never perform erroneous, externally visible operations due to updates made by the privatizer. RTM addresses challenge (2) by means of immediate aborts: by definition, privatization must logically conflict (on *some* datum somewhere) with any ongoing transaction that would be imperiled by subsequent private writes. Before committing, the privatizing transaction is therefore guaranteed to force such transactions to abort, either explicitly or by invalidating words they have *ALoad*ed.

Regarding challenge (1), we provide access to privatized objects via "smart pointers" that hide the extra indirection shown in Figure 2. Initialization of the smart pointer provides a hook that allows RTM to make sure that the object is "clean"—that all committed updates are in place. We could make the initialization operation faster in the common (no cleanup required) case by requiring a thread to perform an explicit "transactional fence" operation before using privatized data; we are currently exploring this option [30].

3.7 RTM-Lite

While both AOU and PDI can improve the performance of TM, AOU is a much smaller change to existing cache designs—an AOU-capable processor can, in fact, be completely compatible with existing bus interfaces and protocols, where a PDI-capable processor cannot. An analysis of overheads in software TM also suggested that AOU alone could yield significant performance gains. We therefore designed a system that relies only on this mechanism.

Like a fast-path RTM transaction, an RTM-Lite transaction *ALoad*s the headers of the objects it reads. It does not add itself to Overflow Reader lists. Since *TStore* is not available, it must clone every acquired object. At the same time, it never has to worry about in-place updates, so immediate aborts are not required. This avoids some complexity in the run-time system: the alert handler simply sets the transaction descriptor to `ABORTED` and returns. A transaction checks its status on every API call, but this takes constant time: in comparison to RSTM, validation requires neither a cache-miss-inducing change to a visible reader list nor an $O(n)$ check of n previously-opened objects.

RTM-Lite transactions actually resemble RSTM transactions (with invisible reads) more closely than they resemble either fast-path or overflow transactions in RTM. We therefore created the RTM-Lite code base by adding *ALoad*s to RSTM and removing validation, rather than by removing in-place update from RTM. As a result, RTM-Lite shares some additional, incidental similarities to RSTM: Instead of using a *Wide-CAS* to update multiple header fields atomically, RTM-Lite moves several fields into the data object and requires an extra level of indirection to read an object whose owner has aborted.[3] Instead of using serial numbers to recognize re-used descriptors, RTM-Lite requires both committed *and* aborted transactions to clean up Owner pointers in acquired objects.

Every RTM-Lite transaction keeps an estimate of the number of lines it can safely *ALoad*. If it opens more objects than this, it keeps a list of the extra objects and validates them incrementally, as RSTM does. If it suffers a capacity/conflict alert, it reduces its space estimate, aborts, and restarts. On a context switch, RTM-Lite transactions abort and restart as RSTM transactions.

As noted in Section 1, RTM-Lite shares much of its design philosophy with the Intel HASTM system [21]. From a hardware perspective, the principal distinction between the systems is that AOU provides asynchronous alerts when designated cache lines are evicted, while HASTM increments a saturating counter. While the counter may be simpler to implement than an asynchronous alert, it does not support immediate aborts. These help us ensure correctness in the presence of privatization or in-place updates. They also enable several useful applications other than transactional memory [25, 28] (though simultaneous use for more than one purpose would require software convention/agreement).

By *ALoad*ing object headers in fast-path transactions and combining visible readers and *ALoad*ed transaction descriptors in overflow transactions, RTM-Lite inexpensively ensures that a transaction never sees inconsistent memory. Without immediate aborts, a doomed transaction might see inconsistencies between or, given privatization or in-place updates, even within objects. Incremental validation at open time suffices to catch inter-object inconsistency, and is all one really needs in clone-based systems, where committed data is immutable [8,27]. One could protect against intra-object inconsistency by performing incremental validation on every read, but even with HASTM hardware, this would be unacceptably expensive for transactions whose read set overflows the cache. Alternatively, one could tolerate inconsistency via sandboxing, but this appears to be possible only for typesafe, managed languages, and then only with extensive compiler support. (Among other things, one must ensure that a "confused" transaction can never store to a nontransactional location, e.g. through a garbage pointer.)

4. EVALUATION

We present experimental results to evaluate our three main claims: (1) RTM hardware can be effectively used to speed a software TM system, (2) policy flexibility is important, and (3) the hybrid design permits a heterogeneous mix of fast-path and overflow transactions without impeding throughput.

[2] A transaction is said to privatize object X if it modifies program state in such a way that subsequent successful transactions will not access X. A "private" object may be shared by more than one thread if those threads use locks or some other non-transactional mechanism to avoid races among themselves.

[3] Indirect access to transactional objects, inherited from RSTM, facilitates nonblocking progress in RTM-Lite, but it is not required in the presence of AOU. We have recently developed a nonblocking successor to RTM-Lite that performs all updates in place [29].

4.1 Evaluation Framework

We evaluate RTM through full system simulation of a 16-way chip multiprocessor (CMP) with private split L1 caches and a shared L2. We use the GEMS/Simics infrastructure [15], a full system functional simulator that faithfully models the SPARC architecture. The instructions specified in Section 2 interface with the TM runtime systems using the standard Simics "magic instruction" mechanism. We implemented the TMESI protocol and alert-on-update mechanism using the SLICC [15] framework to encode all the stable and transient states in the system.

16-way CMP, Private L1, Shared L2	
Processor Cores	16 1.2GHz in-order, single issue, non-memory IPC=1
L1 Cache	64kB 4-way split, 64-byte blocks, 1 cycle uncontended latency, 32 entry victim buffer
L2 Cache	8MB, 8-way unified, 4 banks, 64-byte blocks, 20 cycle uncontended latency
Memory	2GB, 100 cycle latency
Interconnect	4-ary ordered tree, 1 cycle link latency, 64-byte links

Table 3: Target System Parameters

We employ GEMS's network model for interconnect and switch contention, using the parameters in Table 3. Simics allows us to run an unmodified Solaris 9 kernel on our target system with the "user-mode-change" and "exception-handler" interface enabling us to trap user-kernel mode crossings. On crossings, we suspend the current transaction context and allow the OS to handle TLB misses, register-window overflow, and other kernel activities required by an active user context in the midst of a transaction. On transfer back from the kernel we deliver any alert signals received during the kernel routine, triggering the alert handler as needed. On context switches, we simulate the execution of the simple software handlers described in Section 3.5.

HashTable: Transactions use a hash table with 256 buckets and overflow chains to lookup, insert, or delete a value in the range $0 \ldots 255$ with equal probability. At steady state, the table is 50% full.
RBTree: In the red-black tree (RBTree) benchmark, transactions attempt to insert, remove, or delete values in the range $0 \ldots 4095$ with equal probability. At steady state there are about 2048 objects, with about half of the values stored in leaves.
RBTree-Large: This version of the RBTree benchmark uses 256-byte tree nodes to increase copying overheads. Transactions only modify a small percent of the fields of the node.
LFUCache: LFUCache uses a large (2048) array based index and a smaller (255 entry) priority queue to track the most frequently accessed pages in a simulated web cache. When re-heapifying the queue, transactions always swap a value-one node with a value-one child; this induces hysteresis and gives each page a chance to accumulate cache hits. Pages to be accessed are randomly chosen using a Zipf distribution: $p(i) \propto \Sigma_{0<j\leq i} j^{-2}$.
LinkedList-Release: In the LinkedList-Release benchmark, *early release* is used to minimize read-set size while performing inserts, lookups, and deletes into a sorted, singly-linked list holding values in the range $0 \ldots 255$.
RandomGraph The RandomGraph benchmark requires transactions to insert or delete vertices in an undirected graph represented with adjacency lists. Edges in the graph are chosen at random, with each new vertex initially having up to 4 randomly selected neighbors.

Table 4: Workload Description

We consider the six benchmarks listed in Figure 4, designed to stress different aspects of software TM. In all the benchmarks, we execute a fixed number of transactions in single-thread mode to ensure that the data structures are in steady state. We then execute a fixed number of transactions concurrently in order to evaluate throughput and scalability.

4.2 Runtime Systems Evaluated

We evaluate each benchmark with two RTM configurations. RTM-F always executes fast-path transactions to extract maximum benefit from the hardware; RTM-O always executes overflow transactions to demonstrate worst-case throughput. We also compare RTM to RSTM and to the RTM-Lite runtime described in Section 3.7. As a baseline best-case single-thread execution, we compare against a coarse-grain locking library (CGL), which enforces mutual exclusion by mapping the `BEGIN_` and `END_TRANSACTION` macros to acquisition and release of a single coarse-grain test-and-test-and-set lock.

Since RTM-Lite uses AOU to eliminate the incremental validation of RSTM's invisible reads, we configured RSTM to use invisible reads for all experiments. This provides a clearer evaluation of the benefit of AOU, although it complicates comparison to RTM-O. In separate experiments, we compared visible and invisible read throughput in RSTM on a SunFire T1000 (CMP) and SunFire 6800 (16-way SMP). We found that visible reads scaled dramatically worse on the SMP, due to the cost of cache misses induced by modifying reader lists, but offered much higher single-thread performance. On the CMP, visible reads scaled at roughly the same rate as invisible reads, with the same performance spike at one thread. Since we simulate a CMP, we expect our choice of read strategy in RSTM to have little effect on its relative performance.

To ensure a fair comparison, we use the same benchmark code, memory manager, and contention manager (Polka [23]) in all systems. To avoid the need for a hook on every load and store, however, we modify the memory manager to segregate the heap and to place shared object payloads at high addresses (metadata remains at low addresses). The simulator then interprets memory operations on high addresses as *TLoad*s and *TStore*s.

4.3 Throughput and Latency

Figure 3 presents normalized throughput (transactions/sec.) for all benchmark and runtime combinations. We use eager conflict detection, and normalize results to single-thread CGL performance. Figure 4 presents a breakdown of transaction latency at 1 thread and 8 threads. **App Tx** represents time spent in user-provided code between the `BEGIN_` and `END_TRANSACTION` macros; time in user code outside these macros is **App Non-Tx**. **Validation** records any time spent by the runtime explicitly validating its read set; **Copy** is the time spent cloning objects; **MM** is memory management; **CM** is contention management. Fine-grain metadata and bookkeeping operations are aggregated in **Bookkeeping**. For single-thread runs, the time spent in statistics collection for contention management is merged into bookkeeping; for multi-thread runs, **Abort** is the sum of all costs in aborted transactions. Our results highlight: 1) overheads of RTM relative to CGL on a single thread; 2) scalability of RTM relative to RSTM across all thread levels; and 3) the relative benefits of AOU alone vs. AOU + PDI.

On a single thread, RTM-Lite leverages AOU to achieve up to a 5× speedup over RSTM. RTM-F exploits PDI and a the shorter code path to attain a maximum speedup of 8.7× on RandomGraph and a geometric-mean speedup of 3.5× across all the benchmarks. This brings the throughput of RTM-F to 35-50% of CGL through-

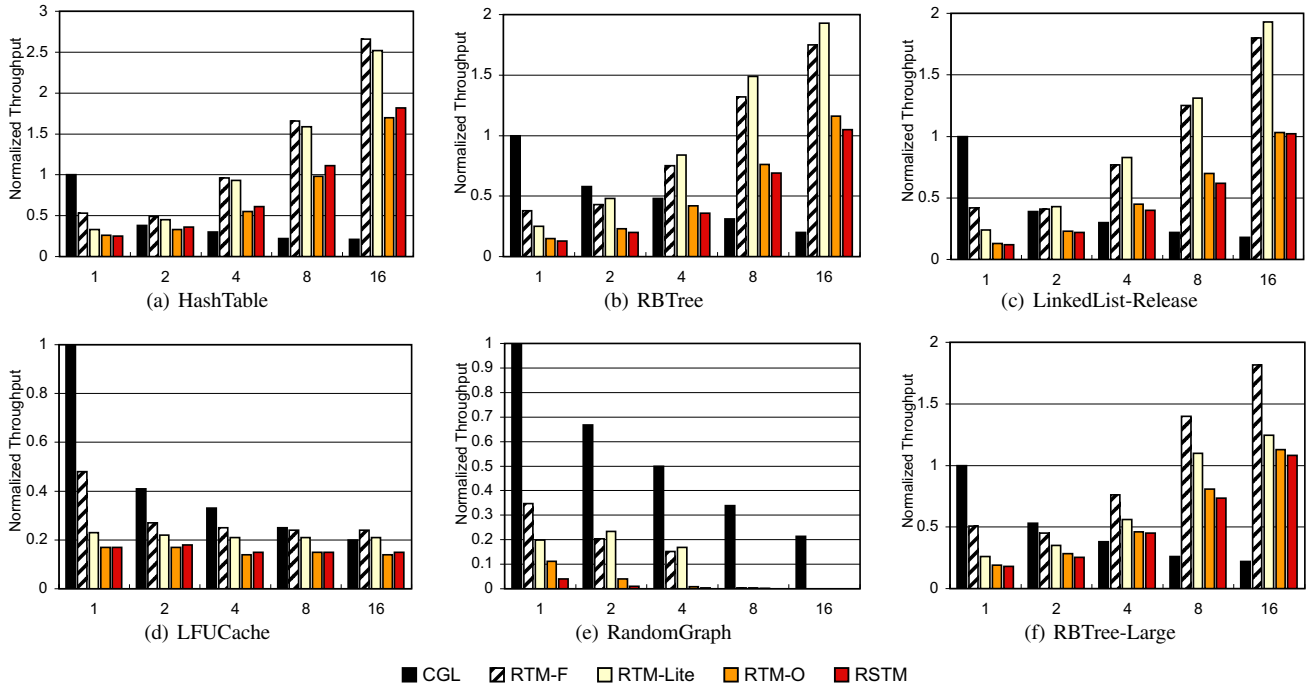


Figure 3: Throughput (transactions/10^6 cycles), normalized to 1-thread CGL. X-axis specifies the number of threads.

put on a single thread. Figure 4 shows that bookkeeping is the dominant overhead in RTM-F relative to CGL. One source of overhead is RTM's use of indirection to access an object. This in turn stems from our choice of metadata organization (Figure 2). Additional improvements in single-thread performance might be realized by eliminating indirection; we leave this for future work.

Despite single-thread performance loss, RTM shows better scalability and performance than CGL for benchmarks with concurrency. RTM-F, RTM-O, RTM-Lite, and RSTM scale similarly across all benchmarks. As the breakdowns show, RTM and RTM-Lite appear to successfully leverage *ALoad* to eliminate RSTM's validation overhead. RTM-F also eliminates copying costs. Due to the small object sizes in benchmarks other than RBTree-Large, this gain is sometimes compensated for by differences in metadata structure and corresponding bookkeeping overhead. Similar analysis at 8 threads (Figure 4) continues to show these benefits, although increased network contention and cache misses, as well as limited concurrency in some benchmarks, cause increased latency per transaction. In RSTM, as the number of threads is increased, validation and copying costs increase due to network contention and cache misses. For example, with HashTable, at 8 threads, the validation cost increases by 2.2× and copying cost increases by 2× when compared to a single thread. RTM-Lite and RTM-F therefore show improved scalability over RSTM.

HashTable exhibits embarrassing parallelism since transactions are short (at most 2 objects read and 1 written) and conflicts are rare. Even RSTM is able to scale to a higher throughput than single-thread CGL. However, these properties prevent the hardware from offering much additional benefit. In single thread mode, the cost of copying is 4.3% of transaction latency in RSTM. Since the read set is small the cost of validation is 16.8%. RTM-Lite eliminates the validation overhead and reduces bookkeeping to improve transaction latency by 23%. In single-thread mode, RTM-F minimizes bookkeeping over RTM-Lite by 2.2× due to single-thread optimizations. Memory management and bookkeeping account for 52% of RTM-F's transaction latency, resulting in RTM-F achieving 53% of CGL's throughput. Since conflicts are rare, even at 8 threads, aborts account for only 4% of the total time. At 16 threads, RTM-F and RTM-Lite attain 2.67× and 2.52× the throughput of single-thread CGL, respectively.

For LinkedList-Release, the use of early release keeps conflicts low. However, LinkedList-Release has a high cost for metadata manipulation and bookkeeping (on average 64 objects are read and 62 released). Together, they account for 88% of the transaction latency of an RSTM transaction. RTM-Lite eliminates the validation overhead, resulting in 24% of the throughput of CGL. In addition, RTM-F lowers RSTM's bookkeeping overheads by 2.8× to attain 42% the throughput of CGL on a single thread. Even at high thread levels, aborts account for only 5% of the latency across all the TM systems. This enables all the TM systems to demonstrate good scalability. At higher thread levels, RSTM suffers an increase in validation and bookkeeping costs. RTM-F and RTM-Lite both exploit the hardware to eliminate the validation overhead of RSTM at all thread levels and attain ∼2× the throughput of single-thread CGL at 16 threads. RTM-Lite performs slightly better than RTM-F at more than 1 thread. RTM-F has 11% higher bookkeeping compared to RTM-Lite at $>$ 1 thread. This increase in bookkeeping outweighs the benefits obtained from eliminating the copying.

Tree rebalancing in RBTree gives rise to conflicts, with read and write sets above 10 and 3 objects, respectively. By deferring to transactions that have invested more work, and by backing off before retrying, the Polka [23] contention manager keeps aborts to within 5% of the total transactions committed. At 8 threads, transactions spend ∼10% of the time in contention management. All TMs scale well. As shown in Figure 4, validation is a significant overhead in RSTM transactions (40% of transaction latency). Copying and memory management costs are negligible. Both RTM-F and RTM-Lite eliminate the validation cost. At 16 threads, RTM-F attains 1.75× the throughput of single-thread CGL.

RBTree-Large scales much as RBTree, but RTM-F is able to

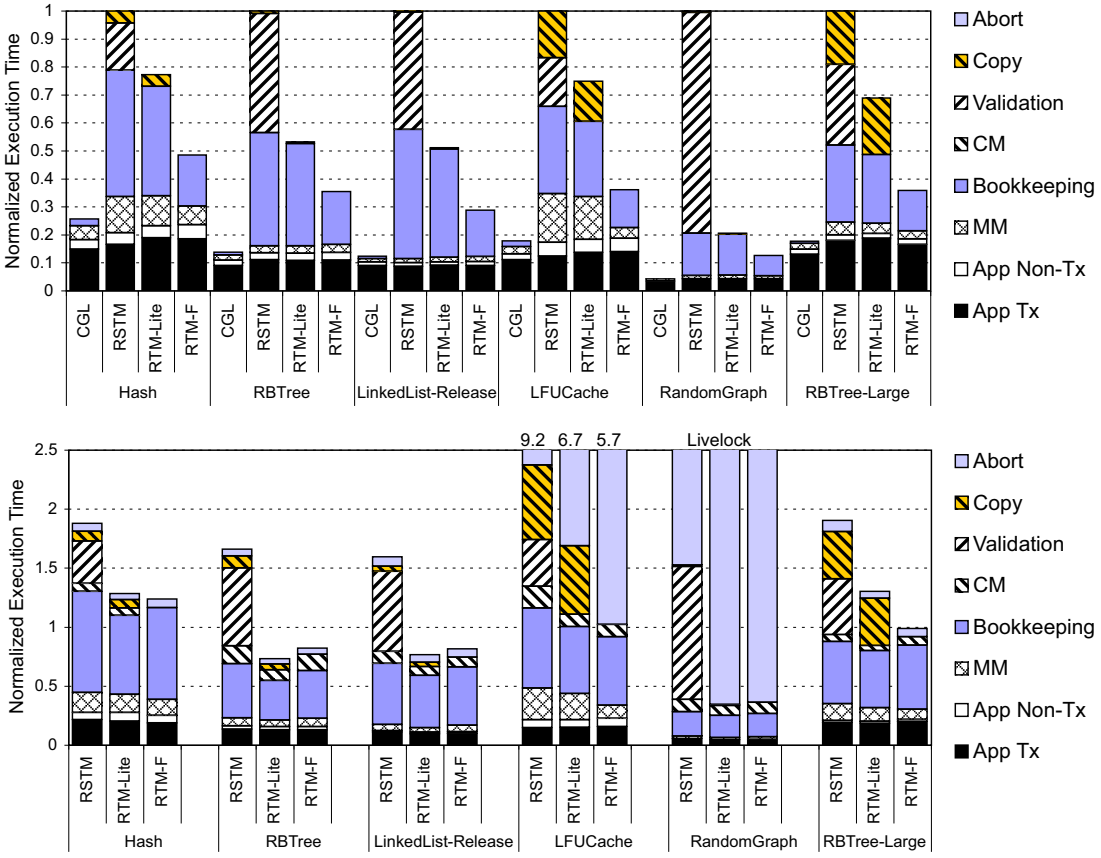


Figure 4: Breakdown of per transaction latency for 1 thread (top) and 8 threads (bottom). All results are normalized to 1-thread RSTM.

leverage *TLoad* and *TStore* to eliminate long copy operations. RTM-Lite reduces transaction latency by 31% over RSTM by eliminating validation. RTM-F reduces latency another 19% by eliminating the copy overhead. On a single thread, reduced bookkeeping allows RTM-F to get within 50% of the throughput of CGL. Copy operations are a dominant overhead at higher thread levels and RTM-F achieves 1.8× the throughput of single-thread CGL.

LFUCache has little concurrency due to the Zipf distribution used to choose which page to access. RSTM and RTM-Lite flatline as threads are added. RTM-F's throughput degrades from 1 to 2 threads (due to the increased complexity of the code path when there are multiple threads) and then flat-lines. On a single thread, RTM-F eliminates copying and validation, and reduces bookkeeping, resulting in 2.6× the throughput of RSTM. With more threads, LFUCache transaction latency is dominated by aborts. At 8 threads, RSTM transactions take 9.2× as long as they do on a single thread, and aborts account for 75% of total time. Commit latency in a transaction increases by 2.4× compared to a single thread. *ALoad* allows RTM-F and RTM-Lite to outperform RSTM by factors of 1.6 and 1.4 at 16 threads, respectively, but CGL's peak throughput (at a single thread) is still 4× higher than RTM-F.

Transactions in RandomGraph are complex; they read hundreds of objects, write tens of objects, and conflict with high probability. Validation is expensive and aborts are frequent. In RSTM, validation dominates single thread latency, contributing to 79% of overall execution time. RSTM has 25× lower throughput than CGL. Leveraging *ALoad* to eliminate validation enables RTM-Lite and RTM-F to improve over RSTM, resulting in 20% and 35% the throughput of 1-thread CGL, respectively. RTM-Lite demonstrates 5× higher throughput than RSTM. Similarly, RTM-O's use of a visible reader list enables it to avoid validation and outperform RSTM by 2.8×, although the two perform comparably when RSTM is configured to use visible readers. RTM-F improves throughput by a factor of 8.7× compared to RSTM.

When there is any concurrency, the choice of eager acquire causes all TMs to livelock in RandomGraph with the Polka contention manager. In the next section, we demonstrate the use of lazy conflict detection to avoid this pathological case.[4]

4.4 Advantages of Policy Flexibility

In order to study the advantages of policy flexibility, we evaluated RTM-F under two different conflict detection policies, eager and lazy. Figure 5 presents the results for HashTable, RBTree, LFUCache, and RandomGraph using 16 threads. For RBTree-Large the eager/lazy tradeoff demonstrates behavior similar to RBTree and for LinkedList-Release the performance variation between eager and lazy was 2% in favor of lazy at 16 threads.

Neither lazy nor eager is a clear winner, demonstrating the need for policy flexibility. Even in applications like HashTable and RBTree, which scale well with increasing thread levels, the choice of conflict detection policy has a measurable effect on performance,

[4]We have tagged the top of the breakdown plots for 8 threads as "livelock" since almost 99% of transaction time is spent in aborts. This livelocking behavior can be avoided even with eager acquire by using a *Greedy* contention manager [6], modified to support invisible reads. Using Greedy, all TMs flat-line in RandomGraph as threads are added.

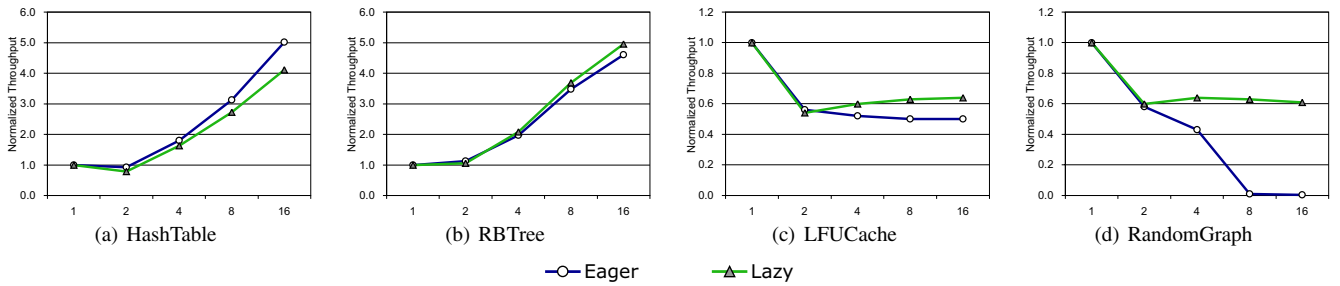


Figure 5: Eager vs. lazy conflict detection comparison, for RTM-F transactions. Throughput normalized to RTM Eager, 1 thread. X-axis specifies the number of threads

given different access patterns. Lazy conflict detection requires non-trivial software overhead: the runtime must remember written objects and acquire ownership just prior to commit time. This overhead may outweigh the benefits of increased read-write concurrency and livelock freedom, especially if the application scales well. On a single thread there is no appreciable difference between eager and lazy acquire: single-thread mode optimizations eliminate both header *ALoad*s and acquire operations, leaving little difference between the code paths. Comparing single thread throughput to 2 threads, we notice that there is not much scalability for benchmarks that exhibit parallelism (ie. HashTable and RBTree). This is due to the difference in code paths between the single threaded and multithreaded configurations. For benchmarks with little parallelism (LFUCache and RandomGraph), performance degrades by ∼45% from 1 to 2 threads for both eager and lazy acquire.

For HashTable, where conflicts are rare, the extra bookkeeping overhead of lazy relative to eager acquire results in performance degradation of 11%, 13%, and 21%, respectively, at 4, 8, and 16 threads. However, while RBTree shows a slight degradation with lazy acquire at low thread counts (> 1), at higher thread counts lazy acquire enables RBTree to scale slightly better, achieving an 11% speedup at 16 threads.

In LFUCache, as in RBTree, lazy transactions are slower than eager at 2 threads due to extra bookkeeping. Since LFUCache admits no concurrency, eager acquire hurts performance at higher thread levels. As soon as an eager transaction accesses and acquires an object, it is vulnerable to conflicts with other threads. Since the likelihood of another thread trying to use the same object is high, increased concurrency decreases the likelihood of that transaction committing, causing performance to flatten beyond 2 threads. In contrast, lazy acquire actually *improves* performance slightly. We expect that lazy acquire would not degrade due to contention; conflicts are only visible at the point where one transaction attempts to commit, and at that commit point conflicts are usually only realized between writers on the same object. Since the conflicting transactions are both about to commit, the likelihood of the conflict "winner" ultimately failing is low. Despite LFUCache's lack of concurrency, lazy acquire's ability to overlap a thread's transactional work with another thread's non-transactional work improves performance over eager by 28% at 16 threads.

On RandomGraph, we had earlier noted that at high thread levels all TM systems livelocked under eager acquire. RandomGraph transactions usually `open_RW` at least one highly contended object early, and then continue to read and write multiple objects. The average RTM-F transaction latency in RandomGraph is 19× longer than that of HashTable. Since transactions run for tens of thousands of instructions, the likelihood of another transaction detecting and winning a conflict on the contended object is high under eager acquire, resulting in potentially cascading aborts. With lazy acquire, the conflict window is narrowed since conflicts are only detected at the very end of transactions. The probability of a conflict winner committing is also higher in lazy acquire compared to eager. Figure 5 shows that lazy acquire avoids the pathological livelock resulting from eager conflict detection.

4.5 Interaction of Fast-Path and Overflow Transactions

The previous subsections analyzed the performance of RTM fast-path (RTM-F) transactions. Figure 6 presents average transaction latency as the fraction of fast-path transactions is varied from 0–100%, normalized to latency in the all-fast-path case. The figure shows a linear increase in latency as the percentage of overflow transactions is increased, with the fraction of time spent in overflow mode being directly proportional to this percentage. Overflow transactions do not block or significantly affect the performance of fast-path transactions when executed concurrently.

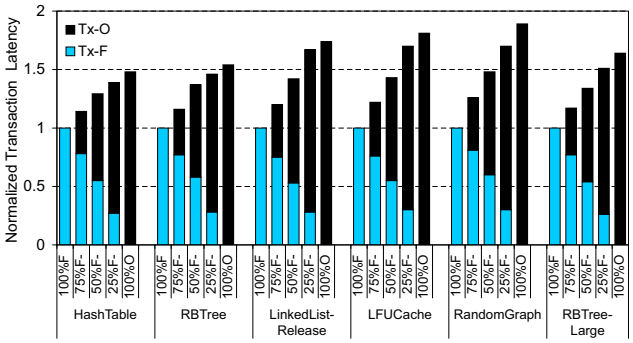


Figure 6: Breakdown of time spent in fast-path and in overflow mode, normalized to the all-fast-path execution (16 threads).

5. CONCLUSIONS AND FUTURE WORK

We have described a transactional memory system, RTM, that uses a pair of hardware mechanisms to accelerate software transactions: *alert-on-update* (AOU) provides fast event-based communication for conflict detection; *programmable data isolation* (PDI) allows a processor to hide speculative writes from other processors and to continue to use speculatively read and written lines despite potentially conflicting writes on other processors.

RTM's fast-path (fully hardware-supported) transactions require cache space only for speculative writes; lines that have only been

read can safely be evicted, as can nontransactional data. Transactions that nonetheless overflow hardware resources fall back gracefully to software, and interoperate smoothly with fast-path transactions. All transactions employ a software contention manager, enabling the use of adaptive or application-specific policies.

Our results show that RTM outperforms RSTM by an average of 3.5× on a single thread and 2× on 16 threads. The simpler RTM-Lite system, which relies on AOU but not PDI, is effective at eliminating validation overhead, but loses to RTM for transactions that modify large objects. Echoing the findings of previous software TM studies, we find significant performance differences between eager and lazy conflict detection, with neither outperforming the other in all cases; this supports the need for policy flexibility.

In future work, we plan to explore a variety of topics, including performance sensitivity to processor organization parameters; simplified protocols (without transactional loads); implementations for other coherence protocols (e.g., MOESI or directory-based); additional hardware assists; nested transactions; gradual fall-back to software, with ongoing use of whatever fits in cache; other styles of RTM software (e.g., word-based, blocking, and/or indirection-free); context identifiers for transactions implemented at a shared level of cache; and more realistic applications.

6. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers and our shepherd, Ravi Rajwar, for suggestions and feedback that helped to improve this paper. Our thanks as well to Virtutech AB for their support of Simics, and to the Wisconsin Multifacet group for their support of GEMS.

7. REFERENCES

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded Transactional Memory. In *Proc. of the 11th Intl. Symp. on High Performance Computer Architecture*, San Francisco, CA, Feb. 2005.

[2] C. Blundell, E. C. Lewis, and M. M. K. Martin. Subtleties of Transactional Memory Atomicity Semantics. *ACM SIGARCH Computer Architecture News*, 5(2), Nov. 2006.

[3] W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. V. Biesbrouck, G. Pokam, B. Calder, and O. Colavin. Unbounded Page-Based Transactional Memory. In *Proc. of the 12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 2006.

[4] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid Transactional Memory. In *Proc. of the 12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 2006.

[5] K. Fraser and T. Harris. Concurrent Programming Without Locks. Submitted for publication, 2004. Available as `research.microsoft.com/~tharris/drafts/cpwl-submission.pdf`.

[6] R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic Contention Management in SXM. In *Proc. of the 19th Intl. Symp. on Distributed Computing*, Cracow, Poland, Sept. 2005.

[7] L. Hammond, V. Wong, M. Chen, B. Hertzberg, B. Carlstrom, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In *Proc. of the 31st Intl. Symp. on Computer Architecture*, München, Germany, June 2004.

[8] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software Transactional Memory for Dynamic-sized Data Structures. In *Proc. of the 22nd ACM Symp. on Principles of Distributed Computing*, Boston, MA, July 2003.

[9] M. Herlihy and J. E. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proc. of the 20th Intl. Symp. on Computer Architecture*, San Diego, CA, May 1993. Expanded version available as CRL 92/07, DEC Cambridge Research Laboratory, Dec. 1992.

[10] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid Transactional Memory. In *Proc. of the 11th ACM Symp. on Principles and Practice of Parallel Programming*, New York, NY, Mar. 2006.

[11] J. R. Larus and R. Rajwar. *Transactional Memory*, Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2007.

[12] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive Software Transactional Memory. In *Proc. of the 19th Intl. Symp. on Distributed Computing*, Cracow, Poland, Sept. 2005.

[13] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the Overhead of Software Transactional Memory. In *ACM SIGPLAN Workshop on Transactional Computing*, Ottawa, ON, Canada, June 2006. Expanded version available as TR 893, Dept. of Computer Science, Univ. of Rochester, Mar. 2006.

[14] J. F. Martínez and J. Torrellas. Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications. In *Proc. of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 2002.

[15] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. In *ACM SIGARCH Computer Architecture News*, Sept. 2005.

[16] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based Transactional Memory. In *Proc. of the 12th Intl. Symp. on High Performance Computer Architecture*, Austin, TX, Feb. 2006.

[17] J. E. B. Moss. Open Nested Transactions: Semantics and Support. In *Proc. of the 4th IEEE Workshop on Memory Performance Issues*, Austin, TX, Feb. 2006.

[18] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proc. of the 34th Intl. Symp. on Microarchitecture*, Austin, TX, Dec. 2001.

[19] R. Rajwar and J. R. Goodman. Transactional Lock-Free Execution of Lock-Based Programs. In *Proc. of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 2002.

[20] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *Proc. of the 32nd Intl. Symp. on Computer Architecture*, Madison, WI, June 2005.

[21] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural Support for Software Transactional Memory. In *Proc. of the 39th Intl. Symp. on Microarchitecture*, Dec. 2006. Orlando, FL.

[22] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A High Performance Software Transactional Memory System for a Multi-Core Runtime. In *Proc. of the 11th ACM Symp. on Principles and Practice of Parallel Programming*, New York, NY, Mar. 2006.

[23] W. N. Scherer III and M. L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *Proc. of the 24th ACM Symp. on Principles of Distributed Computing*, Las Vegas, NV, July 2005.

[24] N. Shavit and D. Touitou. Software Transactional Memory. *Distributed Computing*, 10(2):99-116, Feb. 1997. Originally presented at the *14th ACM Symp. on Principles of Distributed Computing*, Aug. 1995.

[25] A. Shriraman, M. F. Spear, H. Hossain, S. Dwarkadas, and M. L. Scott. An Integrated Hardware-Software Approach to Flexible Transactional Memory. TR 910, Dept. of Computer Science, Univ. of Rochester, Dec. 2006.

[26] A. Shriraman, V. J. Marathe, S. Dwarkadas, M. L. Scott, D. Eisenstat, C. Heriot, W. N. Scherer III, and M. F. Spear. Hardware Acceleration of Software Transactional Memory. In *ACM SIGPLAN Workshop on Transactional Computing*, Ottawa, ON, Canada, June 2006. Expanded version available as TR 887, Dept. of Computer Science, Univ. of Rochester, Dec. 2005, revised Mar. 2006.

[27] M. F. Spear, V. J. Marathe, W. N. Scherer III, and M. L. Scott. Conflict Detection and Validation Strategies for Software Transactional Memory. In *Proc. of the 20th Intl. Symp. on Distributed Computing*, Stockholm, Sweden, Sept. 2006.

[28] M. F. Spear, A. Shriraman, H. Hossain, S. Dwarkadas, and M. L. Scott. Alert-on-Update: A Communication Aid for Shared Memory Multiprocessors (poster paper). In *Proc. of the 12th ACM Symp. on Principles and Practice of Parallel Programming*, San Jose, CA, Mar. 2007.

[29] M. F. Spear, A. Shriraman, L. Dalessandro, S. Dwarkadas, and M. L. Scott. Nonblocking Transactions Without Indirection Using Alert-on-Update. In *Proc. of the 19th Annual ACM Symp. on Parallelism in Algorithms and Architectures*, San Diego, CA, June 2007.

[30] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization Techniques for Software Transactional Memory. TR 915, Dept. of Computer Science, Univ. of Rochester, Feb. 2007.